

Funciones Hash*

Guido Urdaneta

11 de febrero de 2010

1. Introducción

Una función hash es una función que mapea un dato (posiblemente grande y de tamaño arbitrario) de un conjunto U a un dato pequeño, usualmente un número entero comprendido entre 0 y $M - 1$, el cual puede utilizarse como índice en un arreglo, por ejemplo. Al dato a mapear suele llamársele clave, mensaje u otro nombre dependiendo de la aplicación, mientras que al dato mapeado se le puede llamar valor hash, código hash, o simplemente hash.

$$h : U \longrightarrow \{0, 1, 2, \dots, M - 1\}$$

Las funciones hash tienen muchísimas aplicaciones en computación, siendo la búsqueda una de las más importantes. La estructura de datos fundamental para realizar búsquedas usando funciones hash es la tabla hash, en la cual el valor hash se interpreta como la posición de un arreglo en donde se almacena el valor buscado.

Puesto que las funciones hash pueden mapear elementos de un conjunto más grande a uno más pequeño, es posible que dos o más claves sean mapeadas al mismo valor hash. En este caso se habla de una colisión y se desea que la probabilidad de éstas sea muy baja. Los algoritmos de tabla hash básicamente se enfocan en cómo resolver las colisiones. El enfoque más simple es tener una lista para cada posible valor hash, guardar todas las claves con el mismo valor hash en la lista correspondiente y resolver las colisiones con una búsqueda lineal.

1.1. Propiedades deseadas

1.1.1. Bajo costo

Se desea que el costo computacional de una función hash sea bajo. Por ejemplo, un árbol balanceado permite hacer búsqueda binaria realizando aproximadamente $\log n$ comparaciones. Si el calcular la función hash toma un tiempo mayor que realizar $\log n$ comparaciones, entonces puede que no tenga sentido usar una tabla hash.

*Tomado principalmente de TAOCP Volumen 3 (Knuth) y Wikipedia

1.1.2. Uniformidad

Una función hash debe mapear los datos esperados de entrada de la manera más uniforme posible. La razón es que esta es la manera de minimizar el número de colisiones, ya que mientras más colisiones haya, mayor será el tiempo de ejecución de los algoritmos basados en funciones hash. Por ejemplo, si hay muchas claves mapeadas al mismo valor hash en una tabla hash, las búsquedas degenerarán en búsquedas lineales.

2. Algoritmos para implementar funciones hash

2.1. Función hash trivial

Si la clave es lo suficientemente pequeña, es posible que ésta pueda usarse como valor hash sin hacer modificaciones.

2.2. Hash de una palabra

Para hacer una función hash que mapee de datos cuyo tamaño es menor o igual al de la palabra de la computadora (típicamente n dígitos binarios) al conjunto de los enteros comprendidos entre 0 y $M - 1$ (M sería el número de entradas en una tabla hash), existen dos métodos básicos:

1. *El método de división:* $h(K) = K \bmod M$, donde K es la clave. En este caso se recomienda que M no sea potencia de 2 (ya que el resultado sería tomar los bits menos significativos de la clave ignorando los más significativos) y que tampoco divida a $2^k \pm a$, donde k y a son enteros pequeños. En la práctica, usar números primos que no dividan a $2^k \pm a$ (k y a pequeños) suele dar buenos resultados.
2. *El método de multiplicación:* $h(K) = \lfloor M \left(\left(\frac{A}{w} K \right) \bmod 1 \right) \rfloor$, donde w es 2^n y A se recomienda que sea un entero relativamente primo a w . Knuth sugiere un buen valor para A/w es la proporción áurea $\phi^{-1} = (\sqrt{5} - 1) / 2 \approx 0,6180339887$, ya que en este caso valores sucesivos tenderán a dividir intervalos en del segmento $[0, 1)$ según la proporción áurea. La principal ventaja de este método con respecto al de división es que el valor M no es muy importante.

Nota: la operación $x \bmod 1$ cuando x es un número con parte fraccionaria, produce como resultado la parte fraccionaria. Por ejemplo $5,43 \bmod 1 = 0,43$. La notación $\lfloor x \rfloor$ se refiere a la parte entera de x . Por ejemplo, $\lfloor 7,93 \rfloor = 7$. Si $A/w \approx 0,6180339887$ y $M = 1000$ entonces

$$\begin{aligned} h(3) &= \lfloor 1000((0,6180339887 \times 3) \bmod 1) \rfloor \\ &= \lfloor 1000(1,8541019661 \bmod 1) \rfloor \\ &= \lfloor 1000 \times 0,8541019661 \rfloor \\ &= \lfloor 854,1019661 \rfloor \\ &= 854 \end{aligned}$$

del mismo modo, $h(1) = 618$, $h(2) = 236$, $h(4) = 472$, $h(5) = 90$.

En la práctica, si bien con frecuencia las claves suelen ser más grandes que la palabra del computador, se suele utilizar una función hash que produzca un valor hash del tamaño de la palabra, y luego se aplica uno de estos dos métodos para transformar dicho valor a un valor comprendido entre 0 y $M - 1$.

2.3. Funciones hash para arreglos y otras estructuras de propósito general en programación

Cuando la clave es una cadenas de caracteres (o bytes), la distribución de los datos de entrada suele ser poco uniforme. Por ejemplo, si se trata de cadenas con texto en lenguaje natural (e.g., en español) habrá letras y sílabas que se usan con más frecuencia que otras. En estos casos, se recomienda utilizar funciones hash que dependan de todos los caracteres de la cadena, y que dependan de cada carácter de manera diferente.

El procedimiento usual es combinar todos los caracteres en una sola palabra, y posteriormente usar el método de división o multiplicación para obtener un hash comprendido entre 0 y $M-1$. Una manera de combinar los caracteres es sumarlos o aplicar la función XOR, sin embargo, como estas operaciones son conmutativas, el valor hash producido es el mismo para diferentes permutaciones de los mismos caracteres.

A continuación se presentan varios procedimientos propuestos para combinar los caracteres.

2.3.1. Knott

Knott sugiere que antes de sumar o aplicar XOR, se haga un desplazamiento circular de los datos.

```
hash=0;
for i=0 to n-1 {
    hash = hash + rot(s[i]);
}
return hash;
```

2.3.2. Carter y Wegman

Carter y Wegman proponen calcular el hash de una secuencia s de n caracteres o palabras mediante el uso de n funciones hash independientes:

$$h(s) = (h_0(s_0) + h_1(s_1) + \dots + h_{n-1}(s_{n-1})) \bmod M$$

donde las diferentes h_i pueden ser implementadas mediante arreglos precalculados.

```
hash=0
for i=0 to n-1 {
```

```

        hash = hash+h[i](s[i]); //o h[i][s[i]]
    }
    return hash % M;

```

También puede usarse XOR como alternativa a la suma.

2.3.3. Java hashCode()

El método hashCode en Java se supone que debe retornar una función hash apropiada para los objetos de la clase en la cual se define. Por defecto, se suele retornar la dirección de memoria en la que reside el objeto.

En el caso de las cadenas de caracteres inmutables (tipo String), se utiliza la siguiente función:

$$h(s) = s_0 \times 31^{n-1} + s_1 \times 31^{n-2} \times \dots + s_{n-2} \times 31 + s_{n-1}$$

```

hash = 0;
for i=0 to n-1 {
    hash = hash * 31;
    hash = hash + s[i];
}
return hash;

```

2.3.4. Fowler-Noll-Vo (FNV) Hash

La función hash FNV funciona para encontrar un entero de n bits a partir de una secuencia s de octetos (bytes de 8 bits). La combinación involucra una multiplicación por un número primo (similar al Java hashCode) y luego el uso de la función XOR (en lugar de la suma de Java hashCode). El hash en lugar de comenzar en cero, comienza con un valor especial, y el número primo a utilizar debe tener ciertas propiedades.

```

hash = offset_basis;
for i=0 to n-1 {
    hash = hash * FNV_prime;
    hash = hash XOR s[i];
}
return hash;

```

Alternativamente,

```

hash = offset_basis
for each octet_of_data to be hashed
    hash = hash xor octet_of_data
    hash = hash * FNV_prime

return hash

```

Las multiplicaciones son módulo 2^n donde n es el número de bits del hash. Las operaciones *XOR* son entre el byte a considerar y los ocho bits más bajos del hash.

Parámetros de hash FNV para calcular hashes de 32 y 64 bits:

bits (sin signo)	offset_basis	FNV_prime
32	2166136261	16777619
64	14695981039346656037	1099511628211

Esta función hash ha sido muy usada en una gran variedad de aplicaciones prácticas. Para más información consultar <http://www.isthe.com/chongo/tech/comp/fnv/index.html>.

2.4. Sumas de chequeo

Las sumas de chequeo (checksums) son funciones hash que tienen como propósito principal detectar cambios accidentales en una secuencia de datos que ha sido transmitida o almacenada. La idea es que se transmite el dato junto con el valor hash y el receptor calcula el valor hash de la secuencia recibida y la compara con el valor hash recibido. Si hay una discrepancia, significa que hubo un error en la transmisión y se pueden tomar acciones como pedir una retransmisión.

Nótese que los algoritmos de sumas de chequeo no sirven para detectar modificaciones *intencionales*, es decir, modificaciones introducidas por un atacante que conoce el valor hash deseado, pero que es capaz de enviar un mensaje falso que se mapee al mismo valor hash.

2.4.1. Fletcher-32 y Adler-32

Estas funciones permiten calcular una suma de chequeo de 32 bits dada una secuencia de bytes.

Sea $D = d_0, d_1, d_2, \dots, d_{n-1}$ una secuencia de bytes a la que se quiere calcular una suma de chequeo.

La función Fletcher-32 se puede expresar como:

$$A = d_0 + d_1 + \dots + d_{n-1} \text{ mod } 65535$$

$$B = (d_0) + (d_0 + d_1) + \dots + (d_0 + d_1 + \dots + d_{n-1}) \text{ mod } 65535$$

$$\text{Fletcher-32}(D) = B \times 65536 + A$$

o, equivalentemente, con el siguiente algoritmo:

```

sumA=sumB=0;
for (i=0 to n-1) {

```

```

    sumA = (sumA + d[i]) mod 65535 ;
    sumB = (sumB + sumA) mod 65535;
}
return sumB*65536+sumA;

```

La función Adler-32 se puede expresar como:

$$A = 1 + d_0 + d_1 + \dots + d_n \text{ mod } 65521$$

$$B = (1 + d_0) + (1 + d_0 + d_1) + \dots + (1 + d_0 + d_1 + \dots + d_n) \text{ mod } 65521$$

$$\text{Adler-32}(D) = B \times 65536 + A$$

o, equivalentemente, con el siguiente algoritmo:

```

sumA=sumB=1;
for (i=0 to n-1) {
    sumA = (sumA + d[i]) mod 65521 ;
    sumB = (sumB + sumA) mod 65521;
}
return sumB*65536+sumA;

```

Estas funciones tienen la ventaja de que son fáciles de implementar en software.

Como desventaja tienen que para mensajes cortos la suma de bytes suele ser mucho menor que 65521, razón por la cual los valores hash no se distribuyen uniformemente sobre el espacio de 32 bits. Una alternativa es usar Fletcher-16, que utiliza operaciones módulo $2^8 - 1 = 255$ en lugar de módulo $2^{16} - 1 = 65535$.

2.4.2. Cyclic Redundancy Check (CRC)

En este método la clave es una secuencia de bits $B = b_{n-1}b_{n-2} \dots b_1b_0$ cuyos elementos se interpretan como los coeficientes de un polinomio $B(x) = b_{n-1}x^{n-1} + b_{n-2}x^{n-2} \dots b_1x + b_0$. Esta secuencia se mapea a un valor entre 0 y $M - 1$, donde M es una potencia de 2, por ejemplo $M = 2^m$, tomando el residuo de la división del polinomio $B(x)$ entre un polinomio $P(x) = x^m + p_{m-1}x^{m-1} + \dots + p_0$ de grado m .

Entonces $h(s) = h_{m-1}h_{m-2} \dots h_1h_0$ se construye con los coeficientes del polinomio resultante de la operación $B(x) \text{ mod } P(x)$.

Este tipo de función suele ser más apropiado para una implementación en hardware o microprograma, que en software y ha sido utilizado con mucha frecuencia en la detección de errores de transmisión y almacenamiento de datos.

Existen versiones estandarizadas del algoritmo en las que el polinomio $P(x)$ está especificado. Por ejemplo:

CRC-16-ANSI: $x^{16} + x^{15} + x^2 + 1$, utilizado por USB, Modbus y muchos otros protocolos.

CRC-32-IEEE 802.3: $x^{32} + x^{29} + x^{21} + x^{20} + x^{15} + x^{13} + x^{12} + x^{11} + x^8 + x^7 + x^6 + x^2 + x + 1$, utilizado por MPEG-2, PNG y el programa cksum

2.5. Funciones hash criptográficamente seguras

Las funciones hash criptográficas tienen las siguientes características adicionales a las funciones hash en general:

1. *Resistencia preimagen*: dado un valor hash h , debe ser difícil¹ encontrar un mensaje n tal que $hash(m) = h$.
2. *Segunda resistencia preimagen*: Dado un mensaje m_1 debe ser difícil encontrar un mensaje m_2 tal que $hash(m_1) = hash(m_2)$.
3. *Resistencia a colisiones*: Debe ser difícil encontrar dos mensajes m_1 y m_2 tales que $hash(m_1) = hash(m_2)$
4. Una modificación minúscula (e.g., un bit) en el mensaje original ocasiona cambios en el valor hash comparables a un cambio de cualquier otro tipo. En particular, cualquier cambio en el mensaje original idealmente hace que cada uno de los bits en el valor hash cambie con probabilidad 0.5.

Las funciones hash seguras tienen muchas aplicaciones en aplicaciones relacionadas con seguridad de datos (e.g., firmas digitales y códigos de autenticación de mensajes). También pueden ser usadas como funciones hash de propósito general o como sumas de chequeo.

Algunos ejemplos bien conocidos de algoritmos de función hash criptográficamente seguras son SHA-1, SHA-256, MD5 y Tiger, entre otros.

2.6. Hash perfecto

Se dice que una función hash es perfecta cuando es inyectiva, es decir, que cada dato de entrada se mapea a un valor hash diferente. Desafortunadamente las funciones hash perfectas sólo son útiles cuando las entradas están preestablecidas y se conocen de antemano, como por ejemplo, mapear los nombres de los meses a los enteros del 1 al 12. Si la función hash perfecta mapea los n posibles datos de entrada a un rango consistente en n enteros consecutivos, se dice que es una función hash perfecta y mínima.

3. Tablas Hash

Las tablas hash permiten implementar diccionarios y conjuntos.

Las operaciones fundamentales son:

- insertar(clave, dato) o insertar(clave)
- borrar(clave)

¹Diffícil en este contexto significa "más allá de la capacidad de cualquier adversario"

- buscar(clave)

La diferencia entre un diccionario y un conjunto es que en el diccionario, a la clave se le anexa un dato adicional que no es utilizado por los algoritmos de búsqueda, similar al significado de una palabra en un diccionario tradicional. La operación de buscar en un conjunto devuelve verdadero o falso y un diccionario devuelve el dato asociado o un valor especial que indica que la clave no existe.

El tiempo esperado de búsqueda en una tabla hash es $O(1)$ si la función hash distribuye las claves uniformemente. El peor caso es $\Theta(n)$ y se presenta cuando la función hash produce el mismo valor hash para todas las claves.

Una tabla hash puede verse como una generalización de un arreglo. Con un arreglo se almacena el elemento cuya clave es k en la posición k . El elemento cuya clave es k se puede conseguir accediendo a la k –ésima posición del arreglo. Esto recibe el nombre de direccionamiento directo y es aplicable si podemos asignar un arreglo con una posición para cada posible clave.

Las tablas hash se utilizan cuando no se puede asignar un arreglo con una posición para cada posible clave, lo cual ocurre con frecuencia.

En lugar de almacenar un elemento con clave k en la posición k , se utiliza una función hash h y se almacenará el elemento en la posición $h(k)$. Como hemos visto, normalmente el conjunto de claves tiene una cardinalidad mayor que la del rango de la función hash y por lo tanto es posible que dos claves diferentes tengan asociado el mismo valor hash.

Existen dos métodos para el manejo de colisiones: encadenamiento y direccionamiento abierto. El encadenamiento es con frecuencia la opción más simple y efectiva.

3.1. Resolución de colisiones por encadenamiento

La resolución de colisiones por encadenamiento consiste en utilizar una lista enlazada en cada posición del arreglo, de manera tal que cuando varias claves se deben almacenar en la misma posición simplemente se almacenan todas en la lista enlazada de dicha posición.

La inserción y la eliminación pueden hacerse en tiempo $O(1)$ si la inserción se hace siempre en un extremo de la lista de la lista y para la eliminación se usa un apuntador al nodo de la lista.

La inserción y la eliminación serían en tiempo proporcional a la longitud de la lista si se utiliza una comparación de los valores de las entradas para evitar entradas repetidas.

La búsqueda siempre es proporcional al tamaño de la lista y debe ser lineal.

3.1.1. Análisis

El análisis del tiempo de ejecución se hace en función del *factor de carga* $\alpha = n/m$, donde n es el número de claves en la tabla y m es el tamaño del arreglo (número de listas enlazadas). El factor de carga es el número promedio de elementos por cada lista enlazada.

El peor caso es cuando las n claves están en la misma posición del arreglo y se tiene el equivalente a una lista enlazada de tamaño n . El peor caso del tiempo de búsqueda es $\Theta(n)$ más el tiempo requerido para calcular la función hash.

El caso promedio depende de cuan bien la función hash distribuye las claves entre las posiciones del arreglo.

Para realizar el análisis de caso promedio realizaremos las siguientes consideraciones:

1. Se asume **dispersión uniforme simple**: todo elemento tiene la misma probabilidad de caer en cualquiera de las m posiciones.
2. Para $j = 0, 1, \dots, M - 1$, denotamos la longitud de la lista $T[j]$ como n_j . Entonces $n = n_0 + n_1 + \dots + n_{M-1}$.
3. El valor promedio de n_j es $\alpha = n/M$.
4. Se asume que la función hash se puede calcular en tiempo $O(1)$, de manera tal que el tiempo requerido para buscar el elemento con clave k depende de la longitud $n_{h(k)}$ de la lista $T[h(k)]$.

Consideramos dos casos:

1. Si la tabla hash no contiene elementos con la clave k , entonces la búsqueda falla.
2. Si la tabla contiene un elemento con la clave k entonces la búsqueda es exitosa.

Búsqueda fallida *Teorema:* Una búsqueda fallida requiere tiempo $O(1 + \alpha)$

Demostración:

Una búsqueda fallida de la clave k implica recorrer hasta el final la lista $T[h(k)]$. Esta lista tiene una longitud esperada $E[n_{h(k)}] = \alpha$. Por lo tanto, el número esperado de elementos examinados en una búsqueda fallida es α . Si sumamos el tiempo requerido para calcular la función hash, el tiempo total es $\Theta(1 + \alpha)$.

Búsqueda exitosa *Teorema:* Una búsqueda exitosa requiere tiempo $O(1 + \alpha)$

Demostración:

Asumamos que el elemento x buscado es con igual probabilidad cualquiera de los n elementos de la tabla.

El número de elementos examinados en una búsqueda exitosa de x es uno más que el número de elementos que aparecen antes de x en la lista enlazada correspondiente. Estos son los elementos después de x , asumiendo que hacemos las inserciones al principio de la lista. Debemos encontrar el promedio, para los n elementos de la tabla, de cuántos elementos fueron insertados en la lista de x antes de que x fuera insertado.

Para $i = 1, 2, \dots, n$, sea x_i el i -ésimo elemento insertado en la tabla y sea k_i la clave de x_i .

Para todo i y j , definamos X_{ij} como la variable aleatoria que vale 1 si $h(k_i) = h(k_j)$ y 0 de otro modo. Como asumimos dispersión uniforme, tenemos que la probabilidad de que $h(k_i) = h(k_j)$ para valores cualesquiera de i, j es $1/M$ y por lo tanto $E[X_{ij}] = 1/M$.

El número esperado de elementos examinados en una búsqueda exitosa es:

$$\begin{aligned}
 E \left[\frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=1}^{i-1} X_{ij} \right) \right] &= \frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=1}^{i-1} E[X_{ij}] \right) \\
 &= \frac{1}{n} \left(1 + \sum_{j=1}^{i-1} \frac{1}{M} \right) \\
 &= \frac{1}{n} \sum_{i=1}^n \left(1 + \frac{1}{M}(i-1) \right) \\
 &= 1 + \frac{1}{Mn} \sum_{i=1}^n (i-1) \\
 &= 1 + \frac{1}{Mn} \left(\frac{n^2+n}{2} - n \right) \\
 &= 1 + \frac{n-1}{2M} \\
 &= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}
 \end{aligned}$$

Sumando el tiempo para calcular la función hash tendríamos que el tiempo total es $\Theta(2 + \frac{\alpha}{2} - \frac{\alpha}{2n}) = \Theta(1 + \alpha)$.

Si $n = O(M)$, es decir, si el número de elementos en la tabla es proporcional al número de posiciones entonces el tiempo promedio entonces $\alpha = n/M = O(M)/M = O(1)$. Entonces la búsqueda requiere en promedio tiempo constante.

3.2. Direccionamiento abierto

El direccionamiento abierto es una alternativa al encadenamiento para evitar colisiones.

La idea es almacenar directamente en el arreglo las claves. Cada posición contiene una clave o NULO.

Para buscar una clave k el procedimiento es:

1. Calcular $h(k)$ y examinar la posición $h(k)$. Examinar una posición se conoce como sondeo (probe).
2. Si la posición $h(k)$ contiene la clave k , la búsqueda es exitosa. Si contiene NULO, la búsqueda falló.

3. Si la posición $h(k)$ contiene una clave que no es k , se calcula el índice de otra posición a partir de k y del sondeo en el cual se está (los sondeos se enumeran $0, 1, 2, \dots$).
4. Continuar sondeando hasta que se consiga la clave k o NULO.

Se necesita que la secuencia de posiciones sondeadas sea una permutación de los números de las posiciones $\langle 0, 1, \dots, m-1 \rangle$, de manera tal que se examinen todas las posiciones si es necesario y que no se examine ninguna posición más de una vez.

Así, la función hash es

$$h : U \times \{0, 1, \dots, m-1\} \longrightarrow \{0, 1, \dots, m-1\}$$

El requerimiento de que la secuencia de posiciones sea una permutación de $\langle 0, 1, \dots, m-1 \rangle$ es equivalente a requerir que la secuencia de sondeos $\langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$ sea una permutación de $\langle 0, 1, \dots, m-1 \rangle$.

Para insertar, actuar como si se estuviera haciendo una búsqueda e insertar al encontrar la primera posición con NULO.

3.2.1. Métodos para calcular secuencias de sondeo

Lo ideal es dispersión uniforme, que consiste en que cada clave tiene la misma probabilidad de tener las $m!$ permutaciones de $\langle 0, 1, \dots, m-1 \rangle$ como su secuencia de sondeos.

Es difícil implementar la dispersión uniforme, así que en la práctica se aproxima con técnicas que al menos garanticen que la secuencia de sondeos es una permutación de $\langle 0, 1, \dots, m-1 \rangle$.

Ninguna de las técnicas mencionadas a continuación puede generar todas las $m!$ secuencias de sondeo y utilizarán funciones hash auxiliares, las cuales mapean

$$h : U \longrightarrow \{0, 1, \dots, m-1\}$$

Sondeo lineal Dada una función hash auxiliar, la secuencia de sondeos comienza en la posición $h'(k)$ y continúa secuencialmente a lo largo de la tabla (regresando a 0 después de la posición $m-1$).

Dada la clave k y el número de sondeo i ($0 \leq i < m$),

$$h(k, i) = (h'(k) + i) \bmod m$$

El sondeo lineal tiene la desventaja de que tiende a que se formen secuencias largas de posiciones ocupadas y las secuencias ocupadas tienden a crecer, ya que la probabilidad de que se ocupe una posición vacía precedida por i posiciones ocupadas es $(i+1)/m$. El resultado es que los tiempos promedios de búsqueda e inserción se incrementan.

Sondeo cuadrático Al igual que en el sondeo lineal, la secuencia de sondeos comienza en $h'(k)$. A diferencia del sondeo lineal, el sondeo cuadrático salta por la tabla de acuerdo a una función cuadrático sobre número de sondeo:

$$h(k, i) = (h'(k) + c_1i + c_2i^2) \bmod M$$

donde $c_1, c_2 \neq 0$ son constantes que deben ser restringidas a valores que produzcan una permutación de $\langle 0, 1, \dots, m-1 \rangle$.

Si dos claves distintas tienen el mismo valor $h'(k)$ entonces tienen la misma secuencia de sondeos.

Doble hashing Se usan dos funciones hash auxiliares, h_1 y h_2 . h_1 da el sondeo inicial y h_2 da los sondeos restantes.

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod M$$

Se debe asegurar que $h_2(k)$ siempre sea coprimo de M (sin factores comunes excepto 1) para garantizar que la secuencia es una permutación de $\langle 0, 1, \dots, m-1 \rangle$.

Opciones:

- m puede ser una potencia de 2 y que h_2 siempre produzca un impar mayor que 1
- m puede ser primo y que h_2 siempre produzca un número entre 2 y $m-1$

3.2.2. Análisis

Asumiendo dispersión uniforme, que cada clave tiene la misma probabilidad de ser buscada, y que no se borran claves, el número de sondeos de una búsqueda fallida es $1/(1-\alpha)$, al igual que para insertar.

El número esperado de sondeos en una búsqueda exitosa es como máximo $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$

4. Otras aplicaciones

4.1. Filtros de Bloom

Un filtro de Bloom es una estructura de datos probabilística que permite añadir elementos a un conjunto y luego determinar si un elemento pertenece al conjunto. Es posible que se produzcan falsos positivos (que la estructura reporte que un elemento está en el conjunto cuando en realidad no lo está), pero los falsos negativos son imposibles.

Un filtro de Bloom consiste en una tabla de bits $b_0b_1 \dots b_{M-1}$, donde M es razonablemente grande. Para cada clave K_j a incluir en el filtro, calcular k funciones hash independientes $h_1(K_j), h_2(K_j), \dots, h_k(K_j)$ que producen valores hash comprendidos entre 0 y $M-1$ y establecer los correspondientes k bits en 1 (estos k valores no tienen necesariamente que ser distintos entre sí). Así, $b_i = 1$ si y sólo si $h_l(K_j) = i$ para algún j y l .

Para determinar si una clave K está en el conjunto, evaluar primero si $b_{h_l(K)} = 1$ para todos los l tal que $1 \leq l \leq k$; si esto es cierto, entonces es probable que K está en el conjunto (los bits son todos 1 bien porque K había sido añadida al conjunto o porque los bits se establecieron en 1 debido a las adiciones de otras claves), mientras que si algún b_i es cero, se tiene la certeza de que K no está en el conjunto.

4.1.1. Probabilidad de falsos positivos

Suponga que se inserta un elemento. Entonces la probabilidad de que un bit dado se ponga en 1 debido a la aplicación de una de las funciones hash es $1/M$ y la probabilidad de que no se ponga en 1 es $1 - 1/M$. Como son k funciones hash, entonces la probabilidad de un bit determinado no se ponga en 1 es $(1 - \frac{1}{M})^k$. Después de insertar N elementos, la probabilidad de que dicho bit permanezca en 0 es $(1 - \frac{1}{M})^{kN}$ y la probabilidad de que sea 1 es el complemento $1 - (1 - \frac{1}{M})^{kN}$.

Supongamos que se desea realizar una búsqueda de un elemento que no ha sido añadido al conjunto. La probabilidad de obtener un falso positivo sería la probabilidad de que los k bits producidos por las funciones hash sean todos 1, la cual sería

$$\left(1 - \left(1 - \frac{1}{M}\right)^{kN}\right)^k \approx \left(1 - e^{-kN/M}\right)^k$$

Evidentemente, la probabilidad de falsos positivos crece a medida que N (el número de claves insertadas) crece, y decrece a medida que M (el número de bits en el arreglo) decrece.

El valor de k que minimiza la probabilidad de falsos positivos es

$$\frac{M}{N} \ln 2 \approx 0,7 \frac{M}{N}$$

Si se asume que se usará el valor óptimo de k , entonces el número de bits M a utilizar dado el número de elementos a insertar N y la probabilidad deseada de falsos positivos p es

$$M = -\frac{n \ln p}{(\ln 2)^2}$$

4.2. Hash consistente

Normalmente, si se desea modificar el tamaño de una tabla hash (por ejemplo, si se han añadido muchas claves y se desea disminuir el factor de carga, o viceversa) la única alternativa suele ser crear una tabla nueva con el nuevo tamaño deseado y reinsertar todos los elementos en la nueva tabla, lo cual es una operación costosa.

Una alternativa es asignar a cada entrada de la tabla (bucket) un identificador aleatorio entre 0 y $M - 1$ (en el mismo rango de la función hash) y asignar cada clave K al bucket con identificador más cercano a la clave K .

Cuando se desea añadir un bucket deben reasignarse al nuevo bucket las claves cuyo valor hash sea más cercano al identificador del nuevo bucket que al identificador del bucket donde están almacenadas previo a la adición del nuevo bucket.

Cuando se desea eliminar un bucket, cada clave del bucket a eliminar debe ser reasignada bucket con identificador más cercano entre los buckets restantes, similar a una inserción.

El número esperado de claves a mover si hay N claves en total y B buckets es N/B tanto para la adición como para la eliminación de buckets.

El hashing consistente no suele usarse como una estructura de datos en la memoria de un computador, sino en sistemas distribuidos, donde cada bucket es una computadora diferente y mover claves de un nodo a otro es costoso. Un tipo de sistemas distribuidos que utiliza hashing consistente con frecuencia se conoce como tabla hash distribuidas, el cual puede verse como una tabla hash gigante en el que las claves están esparcidas en varias computadoras. Este tipo de sistema es muy usado por aplicaciones P2P para compartir archivos.

4.3. Códigos de autenticación de mensajes

Un código de autenticación de mensajes (MAC) permite autenticar un mensaje. Cuando un usuario A envía un mensaje m a B , A anexa a m un código c que se calcula a partir de m y de una clave secreta compartida con B . Cuando B recibe el mensaje m_r , utiliza la clave secreta compartida para calcular el código de autenticación para el mensaje m_r . Si m_r ha sido adulterado (posiblemente por un adversario) entonces el código de autenticación será diferente al recibido. Esto ocurre porque si el adversario no conoce la clave secreta, será incapaz de sustituir el código de autenticación c por el correspondiente al mensaje adulterado.

Una manera de implementar MACs es utilizando funciones hash criptográficas. En este caso se conoce como HMAC. Posibles maneras de calcular un HMAC son $h(\text{clave}||m)$, $h(m||\text{clave})$ y $h(\text{clave}||m||\text{clave})$ donde $||$ significa concatenación y h es una función hash criptográficamente segura. Se ha demostrado que estos enfoques puede potencialmente ser susceptibles a ataques, entonces lo que se recomienda actualmente es derivar dos claves diferentes a partir de la clave original y utilizar $h(\text{clave}_1||h(\text{clave}_2||m))$.

Una manera común de hacerlo es:

```
clave1 = (byte 0x5c repetido B veces) XOR clave;  
clave2 = (byte 0x36 repetido B veces) XOR clave;  
hmac = hash(clave1 || hash(clave2 || mensaje));
```

donde se asume que la función hash actúa de manera iterativa sobre bloques de B bytes.

Para más detalles, ver <http://www.ietf.org/rfc/rfc2104.txt>.